On the Cover

Delphi 1.0 / Object Pascal

By Andrew J. Wozniewicz

DLLs: Part I

Introducing Dynamic Link Libraries and How to Create Them Using Delphi

Programs running in Windows can share subroutines located in executable files called *dynamic link libraries* (DLLs). Typically, DLLs are used in the context of a large system or application, where many common routines are shared among many programs.

This is the first article in a series that describes how to create and use DLLs with Delphi. In particular, the topics include:

- Creating DLLs using Delphi.
- Declaring import units to automatically make the subroutines in a DLL available to Delphi programs.
- Using subroutines implemented in DLLs.
- Loading DLLs, and resolving subroutine addresses at run time.

First we'll cover basic DLL terminology. Then we'll build some string-handling functions.

Understanding DLLs

A DLL is an executable module containing subroutines that Windows programs can call to perform common, useful tasks. DLLs are one of the more important elements of Windows, and exist primarily to provide services to applications. Windows itself uses DLLs to make Windows subroutines and resources available to Windows applications. It's probably fair to say that DLLs are the fundamental concept of Windows architecture.

Windows itself consists largely of three DLLs named KERNEL, USER, and GDI. These libraries contain the code and data for the Windows application programming interface, or API.

A Delphi application is an executable file that consists of one or more forms or windows. A program interacts with the user at run time and retrieves messages and events from a message queue maintained on its behalf by Windows.

In contrast, DLLs are separate libraries of subroutines, containing procedures and functions that can be called by other programs. However, DLLs are not themselves directly executable and come into play only when another Windows module calls one of the subroutines in the DLL. Specifically, DLLs do not have their own message queues and rely on the flow of messages and events in the application being used.

A Waste of Space

A user can simultaneously run several copies of a program, called *instances*. These instances share the same code in memory. However, if you run several programs containing the same subroutine

library, the same executable code will be cloned in memory. The same routines will reside in memory in multiple copies.

In other words, if you *statically link* the common routines (shared among many programs) as units to each individual program, you end up with identical code that is linked into many executable modules. If you consider that each of the various individual programs can be running simultaneously on the same system, you see that virtually identical code may be loaded unnecessarily into memory in multiple copies. This results in space being wasted instead of used for the needs of applications.

DLLs to the Rescue

The solution to this problem is the DLL, and the process of dynamic linking that eliminates the duplication of common code. *Dynamic linking* is the process that Windows uses to link a subroutine call in one executable module to the actual function in a dynamic library module at run time. Furthermore, if you change any of the subroutines in a common, static subroutine library, you must rebuild all the programs that use the library. However, if you extract the common subroutines and put them into a separate dynamic library, then only the library contains the routines required by all programs.

There is only one copy of a dynamic library loaded in memory at run time, no matter how many programs use it. Additionally, if you change a subroutine in the common library, you need only recompile the library module. The programs that use the library remain intact, unless you have changed the interfaces to the library. DLLs are similar in concept to Object Pascal units. They represent an additional step in the quest for separating concerns among different subsystems, or modules comprising a program. Object Pascal units are only separate until compile/link time, when they are combined to form a single executable file.

DLLs remain separate even when an application is deployed. Therefore, DLLs are more autonomous than Object Pascal units, and they use the operating system's (Windows) facilities rather than the language/compiler facilities to achieve modularization and integration of separate modules. The integration occurs at run time, late in the development cycle. This is desirable from the perspective of reusability and flexibility.

In addition to letting applications share executable code, you can use DLLs to share other resources, such as data and hardware. Windows fonts, for example, are actually shared text-drawing data, and Windows device drivers actually are special DLLs that enable applications to share hardware resources.

Although a DLL can have any valid file name extension, the standard extension is .DLL. Only libraries with the default extension of .DLL can be loaded automatically by Windows. If the library file has a different extension, it must be loaded explicitly by the program that wants to use it. An application can invoke the standard API **LoadLibrary** function to explicitly load a DLL. We'll see an example of this later.

Dynamic Linking

When a Windows program is loaded into memory, the calls to the Windows API will point to the entry of the subroutines in the appropriate DLLs, which also are loaded into memory. Hence, the executable application module relies on the presence of other supplementary modules. Typically, the program cannot run in the absence of the required external DLLs.

Contrast this with the process of static linking, when the addresses of all subroutines are known beforehand — at compile time. The compiler/linker can appropriately resolve any static references throughout the program and substitute actual addresses for symbolic names used in the source code. Static linking results in a fully self-contained executable file.

Creating Custom DLLs

So far, as you have developed Delphi applications, you have been writing Windows programs. Now it's time to develop DLLs *for use by* programs. Many of the principles you have learned for writing Delphi programs are also applicable to writing Delphi DLLs, but there are some important differences to understand.

As mentioned, DLLs enable you to more economically share code and resources among many applications. Although they are important in the overall architecture of Windows, custom DLLs are not absolutely necessary for every Windows application. After all, you have probably built a fair number of Delphi applications without worrying about DLLs. Although each one of these applications relied on the access to Windows DLLs, you did not have to deal with the details of loading and binding to DLLs. This is about to change.

There are several distinct advantages of DLLs that become increasingly important as your applications become larger, more sophisticated, and complex:

- DLLs enable you to share code and resources among many applications.
- Using DLLs, you can easily customize your application for different purposes.
- DLLs facilitate the development of large, complex applications by allowing a more strict separation of subsystems.
- DLLs enable you to streamline access to data and data-like resources (such as business model infrastructures), and to hardware devices.

Note that you can use DLLs written in other language environments, and you can write custom DLLs in Delphi. The interfaces between a DLL and a program are usually transparent and independent of the implementation language. Therefore, any DLL-capable program can call a subroutine in a Delphi-created DLL — whether that program was written in C, C++, Visual Basic, ObjectPAL, dBASE, etc., or an extension to an off-the-shelf application, including macro-language extensions written in Microsoft Word for Windows or Excel.

Your Delphi programs also can call subroutines in DLLs written in other languages. As mentioned, they do it continually whether or not you are aware of it. Every Delphi program must interact with Windows, and that involves making calls to one, and typically more, of the three main Windows DLLs.

Declaring Libraries

Conceptually, DLLs can be treated simply as more autonomous units. However, from a Pascal programmer's perspective, a DLL source code module bears a striking resemblance to a program module.

A DLL is defined in the same manner as a program, with the reserved word **library** replacing the word **program**:

```
library LibMane;
  < declarations and implementation >
begin
  < optional initialization code >
end.
```

Figure 1 shows an example of a library module. It begins with a header, consisting of the reserved word **library**, followed by the library name and a semicolon. The library implementation follows much of the same rules as the program implementation. You can insert **uses** clauses; declare types, constants, and variables; and declare and implement subroutines and classes — in other words, everything you can do within a program.

```
library StrUtils;

  function UpperCaseStr(S: string): string; export;
    begin
      for I := 1 to Length(S) do
        S[I] := UpCase(S[I]);
      Result := S;
    end;

  exports
    UpperCaseStr;

begin
end.
```

Figure 1: An example of a library module.

Again, similar to a program module, an executable block is enclosed within a pair of **begin** and **end** keywords at the end of a library module. Its role is similar to the role of a **unit** initialization block, rather than that of a program's executable block. It's meant to initialize any local data and objects needed by the library.

The **begin..end** initialization block of a library module may (and often does) remain empty. In a library, these keywords must be present, even if there are no data structures to initialize. Conversely, with units, the initialization block's **begin** keyword can be omitted if there is no initialization code.

Using the exports Clause

One of the main reasons for having a library is to reuse common subroutines from different applications. This implies that you must first implement the subroutines to use them. Fortunately, the way you implement subroutines inside a DLL is not much different from how you implement them in a regular program. In fact, you can use the same source code units that implement subroutines in both programs and DLLs. Subroutines that are made public by listing them in a unit's **interface** section can be used directly in a program by being statically linked to it via the **uses** clause. These subroutines can also be linked into a DLL and made available to a greater number of applications simultaneously by being "exported" from the DLL.

Most importantly, when building dynamically linked collections of subroutines (from the perspective of making the subroutines available outside the library), you must explicitly export those subroutines you intend to be visible outside the library by creating an **exports** clause. The syntax for the **exports** clause, in its simplest form, follows this convention:

```
exports
  Subroutine1,
  Subroutine2,
  Subroutine3,
  ...
  SubroutineN;
```

The **exports** clause begins with the reserved word **exports**, followed by a list of exported subroutines separated by commas. The clause is terminated by a semicolon. Here's an example:

```
exports
  FillStr,
  LTrimStr,
  RTrimStr;
```

Only one **exports** clause may be present in a project. You cannot place an **exports** directive inside a unit belonging to a project; **exports** can only be placed inside the main library or program module.

There may be any number of **exports** clauses in a project's main source file. All the subroutines listed inside all the **exports** clauses will be made visible to other modules "by name," and by other means (we'll discuss this more later). When the module that contains an **exports** clause is compiled, the resulting executable file (.EXE or .DLL) contains special entries that enable Windows to dynamically link the exported subroutines — at run time — to all other applications and modules that may need them.

The key to successfully exporting a subroutine from a DLL is to make the subroutine itself exportable. A subroutine cannot be exported unless you explicitly define it as exportable by placing the **export** directive at the end of the subroutine's declaration heading.

export Directive

The **export** directive makes a subroutine exportable and must be placed after the regular heading in the subroutine's declaration:

```
function Name(<Parameters>): ReturnType; export;
```

or

```
procedure Name(<Parameters>); export;
```

Here are some examples:

```
function StripStr(S: string): string; export;

procedure RefreshBuffers; export;
```

The **export** directive instructs the compiler to create special code at the beginning and end of the compiled routine that enables Windows to use it from another executable module.

In summary, to make a global subroutine inside a DLL project available for use by other Windows executable modules, you must ensure that:

- The subroutine is declared with the **export** directive, i.e. it is *exportable*.
- The subroutine's name is listed inside an **exports** clause of the library module, i.e. it is actually *exported*. A subroutine may be made exportable (with an **export** directive) but not actually exported (with an **exports** clause).

Architecting DLLs

As a matter of principle, when building reusable DLLs you should avoid implementing the subroutines being exported as part of the main library module. There's a temptation to begin creating the implementations of the exported subroutines directly inside the main library module — you could simply start writing code in the main library project file. This approach, however, quickly leads to large and unmanageable library files.

You should separate libraries into units for the same reason you divide programs into separate units. That is, managing smaller units with well-defined interface sections is far easier than keeping track of all the subroutines implemented in one place. Remember that you may be implementing both the subroutines you intend to export from the library, and the auxiliary subroutines for the private use of the library itself that are not exported. By using separate units that group closely related subroutines, you impose an additional layer of organization on the library's code.

An additional benefit to organizing your implementation code in units is that you can either use dynamic linking, or statically link the implementing unit into an application. On some occasions, you may decide that linking the code directly into the application is a better approach, or you'll be content to use the services encapsulated by your DLL. In either case, it's better to have one unit source file that you can use in both scenarios.

Hence, a better approach to organizing a library's code is to have all the routines, both exported and private, implemented in a separate unit, or collection of units. The main library module would have the units it incorporates listed in its **uses** clause, and would implement the **exports** clause.

The following example demonstrates how to organize your subroutines into separate units, yet make them exported from the library.

A Simple Library Example

The best way to illustrate how to build a DLL is to step through a practical example. You'll see that the task of DLL writing has been made reasonably simple by Delphi (much simpler than in more traditional environments). However, you must remember that writing DLLs is a traditional coding exercise and that Delphi 1.0 does not support you in any special way beyond providing the raw capabilities of creating and using DLLs.

Let's build an example DLL that implements several useful string-handling subroutines. Here are the steps:

1. Create a new, blank Delphi project.

2. Close it without saving the default form-unit file, Unit1, which Delphi creates by default. Right-click in the Code Editor and select **Close Page** from the pop-up menu. Answer **No** when you are prompted to save the unit. The Code Editor closes because there are no files to edit.

3. Create a new directory named DLLFIRST. Save the project as DLLFIRST.DPR in the new directory. At this point, your project only consists of the main program file, DLLFIRST.DPR.

4. Select **View | Project Source** from the Delphi menu. The Code Editor reopens and displays the DLLFIRST program file (see Figure 2). Now you'll convert this minimal Delphi program to an equally minimal DLL.

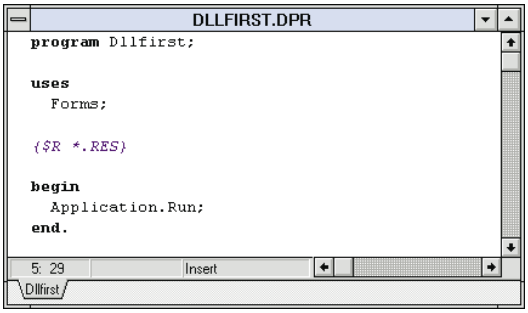


Figure 2: The DLLFIRST project before being modified into a DLL.

5. On the first line of code, replace the reserved word **program** with the word **library**. Then remove:

```
uses
  Forms;

{$R *.RES}
```

Now remove the *Application.Run* statement. Your main source code file is now reduced to:

```
library Dllfirst;

begin
end.
```

6. Congratulations! You've just created the world's simplest DLL. Don't forget to save the project.

7. Select **Compile | Compile** from the Delphi menu. It takes only a fraction of a second to compile this empty library, but you now can verify that instead of generating an executable program file (.EXE) you have succeeded in creating your first dynamic link library: DLLFIRST.DLL.

Of course, this library is not particularly useful in its current state. It does not provide any services that a program might use. Soon, you will develop a set of simple character-string manipulation routines that will be available from this DLL.

An interesting point to note, however, is that the library may be used almost "as is" to store various binary Windows resources, such as bitmaps, icons, and so on. In this context, the DLL is a repository of dynamically installable resources rather than a collection of subroutines. Note that if you had chosen **Run | Run** instead of **Compile | Compile** to compile your DLL, it still would have been compiled, but because it knows that DLLs are not directly executable, Delphi would have displayed the error message "Cannot run a unit or DLL."

If you close the project at this point and attempt to open it later, Delphi opens the file but also displays an error message, "Error in module DllFirst: uses clause missing or incorrect." Delphi is telling you that a **uses** clause is absent from your project file. Delphi needs the **uses** clause to maintain the list of source files belonging to the project, but currently your project consists of the single, main project file. Although there is no true need for the **uses** clause at this point, you have to reintroduce one to placate Delphi's code-generation facilities. So modify the project source to add a **uses** clause for the WinTypes unit:

```
library DllFirst;

uses
  WinTypes;

begin
end.
```

Listing WinTypes in the **uses** clause of the library is safe because it does not introduce any real dependencies. WinTypes is a unit that declares various data types used by the Windows API. It does not implement any code, so you are not adding overhead to your library. Also, you can later delete WinTypes from the **uses** clause entirely. You cannot do that right now, because there are no other units in the project and this would eliminate the **uses** clause altogether. Remember, you cannot leave the reserved word **uses** without unit names listed after it.

Using DLLFirst String-Handling Functions

Now let's implement some string-handling functions inside the DLL. First, let's create a separate unit, named XString, that implements the subroutines. To do this, create a new unit for the project by selecting **File | New Unit** from Delphi's menu. Then save it as XSTRING.PAS by selecting **File | Save File As**.

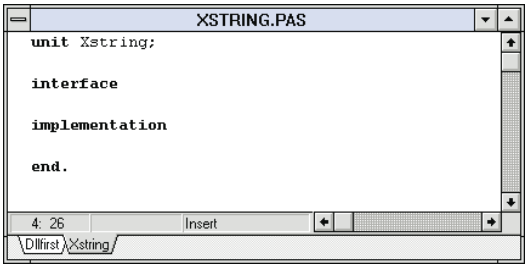


Figure 3: The shell for our sample unit.

The newly-created unit (see Figure 3) is an empty shell.

Now we're ready to provide the actual string-handling routines inside the XString unit. We'll implement the following functions:

- The *FillStr* function returns a string of a single, repeated character value.
- The *UpCaseFirstStr* function capitalizes the first letter of every word of the passed string argument.
- The *LTrimStr* function removes leading blanks from the passed string argument.
- The *RTrimStr* function removes trailing blanks from the passed string argument.
- The *StripStr* function removes all blanks, whether leading, trailing, or embedded, from the passed string argument.

(This month, we'll build and analyze *FillStr* and *UpCaseFirstStr*. We'll move on to the other functions in April.)

In the following sections, note that the declarations of the string-handling subroutines to place inside the **interface** section of the XString unit include the **export** directive, while the implementations no longer repeat and must *not* repeat the directive. As noted earlier, the **export** directive makes these subroutines exportable across the Windows-module (.DLL/.EXE) boundaries.

The *FillStr* Function

The first function to include in your DLL's XString unit is *FillStr*. This classic library function returns a string of the specified length consisting of the indicated character, repeated throughout the entire string a specified number of times.

FillStr is declared as follows:

```
function FillStr(C: Char; N: Byte): string; export;
```

The function takes two parameters: 1) the character to be repeated, `C`; and, 2) the desired length of the string, `N`, or the number of times to repeat the character `C`. Enter this declaration in the **interface** section of the XString unit we just created.

Here's a suggested implementation for the *FillStr* function:

```
function FillStr(C : Char; N : Byte): string;
{ Returns a string with N characters of value C }
begin
  FillChar(Result[1],N,C);
  Result[0] := Chr(N);
end;
```

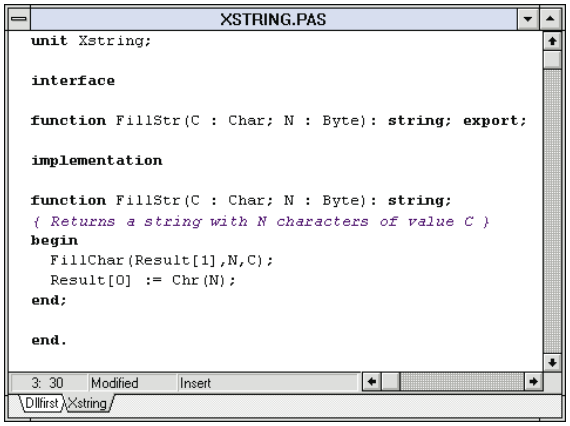


Figure 4: The XString unit with the *FillStr* function implemented.

Enter this code in the **implementation** section of the XString unit. (The entire unit is shown in Figure 4.) The function uses a few tricks to get the job done more efficiently. First, it uses the standard procedure *FillChar* to quickly fill the *Result* string with the appropriate character value. The alternative approach is to set up a **for** loop that counts through the subsequent character positions in the result string and then "stuffs" the character to be repeated in the character cells. *FillChar* accomplishes the same task in a single step.

Second, *FillStr* forces the returned string to be of the appropriate length by explicitly setting the length byte. As you may recall, the length byte of a string variable is the first cell (byte) of the string's character array. *FillStr* sets the length byte directly to ensure that the returned string contains the correct number of characters.

Here's an example of the *FillStr* function in use:

```
var
  S1, S2: string;
begin
  S1 := FillStr(' ',12);
  S2 := FillStr('#',80);
end
```

After the two calls to *FillStr*, S1 will contain a string of 12 blanks, and S2 will contain 80 number-sign (#) characters.

The *UpCaseFirstStr* Function

Another useful addition to the set of standard string-management routines already available in Delphi is the *UpCaseFirstStr* function. It complements the standard *UpperCase* and *LowerCase* string-handling functions from the SysUtils unit. *UpCaseFirstStr* capitalizes the first letter of every word. *UpCaseFirstStr* takes a single string parameter and returns the same string, so it is declared as follows:

```
function UpCaseFirstStr(const S: string): string; export;
```

Enter this declaration in the **interface** section of the XString unit. The body (i.e. the implementation) of *UpCaseFirstStr* mainly consists of a **for** loop that runs through all the characters in the string (see Figure 5). If the character is the first letter of a new word, it is capitalized, if not, it is skipped.

```
function UpCaseFirstStr(const s: string): string;
var
  Index: Byte;
  First: Boolean;
begin
  Result := S;
  First  := True;
  for Index := 1 to Length(S) do
    begin
      if First then
        Result[Index] := UpCase(Result[Index]);
      if Result[Index] = ' ' then
        First := True
      else
        First := False;
    end;
end;
```

Figure 5: The *UpCaseFirstStr* function.

The *First* variable is used to track if the *Index* variable marks the position of the beginning of a word inside string S at any given point. The *First* flag becomes *True* whenever the current *Index* corresponds to the first letter of a word inside the string S. The flag is set with this **if** statement:

```
if Result[Index] = ' ' then
  First := True
else
  First := False;
```

The new value of *First* to be used in the next iteration of the **for** loop is *True* whenever the current location in the string S is occupied by a blank (ASCII 32 decimal or ' '). This way, after a blank character is encountered, any non-blank character that follows is treated as the beginning of a new word.

The following condition determines if the current character should be capitalized:

```
if First then
```

This checks if the *First* flag is set (i.e. whether the previous character in the string was blank) and if so, attempts to capitalize the current character. Note that the *First* flag is initially set to *True* just before the loop begins execution. This implicitly assumes that the first non-blank character in a string also marks the beginning of a new word and forces it to be capitalized. (Remember to always initialize your variables.)

Place the code of the *UpCaseFirstStr* function (see Figure 6) inside the **implementation** section of the XString unit file.

Here's an example of how *UpCaseFirstStr* could be used:

```
var
  S1: string;
begin
  S1 := UpCaseFirstStr('teach yourself delphi in 21 days.');
end
```

```
unit Xstring;

interface

function FillStr(C : Char; N : Byte): string; export;
function UpCaseFirstStr(const s: string): string; export;

implementation

function FillStr(C : Char; N : Byte): string;
begin
  FillChar(Result[1],N,C);
  Result[0] := Chr(N);
end;

function UpCaseFirstStr(const s: string): string;
var
  Index: Byte;
  First: Boolean;
begin
  Result := S;
  First  := True;
  for Index := 1 to Length(S) do
    begin
      if First then
        Result[Index] := UpCase(Result[Index]);
      if Result[Index] = ' ' then
        First := True
      else
        First := False;
    end;
end;

end.
```

Figure 6: The XString unit with the *UpCaseFirstStr* function implemented.

After the assignment statement is executed, the string `S1` contains:

```
'Teach Yourself Delphi in 21 Days.'
```

Until Next Time

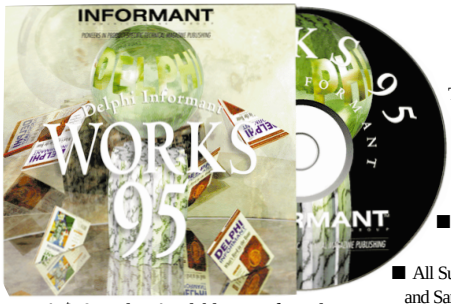
In the next article, we'll implement several more Delphi functions into our sample DLL. These functions include *LTrimStr*, *RTrimStr*, and *StripStr*. We'll also discuss how to export the DLL functions we've created, and how to call them from a program. See you then. Δ

This article was adapted from material for *Teach Yourself Delphi in 21 Days* [SAMS, 1995], by Andrew Wozniewicz

Andrew J. Wozniewicz is president and founder of Optimax Development Corporation (http://www.webcom.com/~optimax), a Chicago-based consultancy specializing in Delphi and Windows custom application development, object-oriented analysis, and design. He has been a consultant since 1987, developing primarily in Pascal, C, and C++. A speaker at international conferences, and an early and vocal advocate of component-based development, he has contributed articles to major computer industry publications. Andrew can be contacted on CompuServe at 75020,3617 and on the Internet at optimax@optidevl.com.

The Must Have Reference Source For The Serious Delphi™ Developer

The Entire Text of Over 60 Technical Articles Appearing in Delphi™ Informant® in 1995



The Delphi™ Informant® Works 1995 CD-ROM Includes:

- Over 60 Technical Articles
- Text and Keyword Search Capability
- Improved Speed and Performance
- All Supporting Code and Sample Files
- 16-Page Color Booklet
- Electronic Version of *Delphi Power Tools* Catalog
- Electronic Version of *Borland C++ Power Tools* Catalog
- Third-Party Add-In Product Demos
- CompuServe Starter Kit with $15 Usage Credit.

A $105 Value Available Now for only

$39.95

California residents add 7¼% Sales Tax, plus $5 shipping & handling for US orders. (International orders add $15 shipping & handling)

Call Now Toll Free 1-800-88-INFORM

1-800-884-6367 Ask for offer # PI95 To order by mail, send check or Money Order to:
Informant Communications Group, Inc.
ATTN: Works CD offer # PI95
10519 E. Stockton Blvd, Suite 142 Elk Grove, CA 95624-9704
or Fax your order to
916-686-8497

Get Informed!

Subscribe to Delphi Informant, The Complete Monthly Guide to Delphi Development and stay ahead of the rapid application development curve.

Order Now and Get One Issue FREE!

For a limited time you can receive the first issue FREE plus 12 additional issues for only $49.95 That's nearly 25% off the yearly cover price!

Each big issue of *Delphi Informant* is packed with Delphi tips, techniques , news, and more!

- Client/Server Application Development
- Delphi Programming
- Using the Borland Database Engine
- Object -Oriented Programming
- Creating Reusable Components
- Product Reviews
- News from the Delphi Community
- Delphi User Group Information

Payment Method...

- ❑ Check (payable to Informant Communications Group)
- ❑ Purchase Order-- Provide Number _____
- ❑ Visa
- ❑ Mastercard
- ❑ American Express

_____
Card Number

_____
Expiration Date

_____
Signature

International rates

Magazine-only
$54.95/year to Canada
$74.95/year to Mexico
$79.95/year to all other countries

Magazine AND Companion Disk Subscriptions
$124.95/year to Canada
$154.95/year to Mexico
$179.95 to all other countries

California Residents add 7¼% sales tax on disk subscription

☐ YES

To order, mail or fax the form below or call (916) 686-6610 Fax: (916) 686-8497

I want to sharpen my Delphi Programming skills. I've checked the subscription plan I'm interested in below

☐ Magazine-Only Subscription Plan...
13 Issues Including One Bonus Issue at $49.95.

☐ Magazine AND Companion Disk Subscription Plan...
13 Issues and Disks Including One Bonus Issue and Disk at $119.95
The Delphi Informant Companion Disk contains source code, support files, examples, utilities, samples, and more!

☐ Delphi Informant Works 1995 CD-ROM $39.95
US residents add $5 shipping and handling. International customers add $15 shipping and handling.

Name _____

Company _____

Address _____

City _____ State _____ Zip Code _____

Country _____ Phone _____

FAX _____ e-Mail _____

PI95